Prefetched Address Translation

Prefetched Address Translation

Artemiy Margaritov
University of Edinburgh
artemiy.margaritov@ed.ac.uk

Dmitrii Ustiugov*
University of Edinburgh
dmitrii.ustiugov@ed.ac.uk

Edouard Bugnion
EPFL
edouard.bugnion@epfl.ch

Boris Grot
University of Edinburgh
boris.grot@ed.ac.uk

ABSTRACT

With explosive growth in dataset sizes and increasing machine memory capacities, per-application memory footprints are commonly reaching into hundreds of GBs. Such huge datasets pressure the TLB, resulting in frequent misses that must be resolved through a page walk – a long-latency pointer chase through multiple levels of the in-memory radix tree-based page table.

Anticipating further growth in dataset sizes and their adverse affect on TLB hit rates, this work seeks to accelerate page walks while fully preserving existing virtual memory abstractions and mechanisms – a must for software compatibility and generality. Our idea is to enable direct indexing into a given level of the page table, thus eliding the need to first fetch pointers from the preceding levels. A key contribution of our work is in showing that this can be done by simply ordering the pages containing the page table in physical memory to match the order of the virtual memory pages they map to. Doing so enables direct indexing into the page table using a base-plus-offset arithmetic.

We introduce Address Translation with Prefetching (ASAP), a new approach for reducing the latency of address translation to a single access to the memory hierarchy. Upon a TLB miss, ASAP launches prefetches to the deeper levels of the page table, bypassing the preceding levels. These prefetches happen concurrently with a conventional page walk, which observes a latency reduction due to prefetching while guaranteeing that only correctly-predicted entries are consumed. ASAP requires minimal extensions to the OS and trivial microarchitectural support. Moreover, ASAP is fully legacy-preserving, requiring no modifications to the existing radix tree-based page table, TLBs and other software and hardware mechanisms for address translation. Our evaluation on a range of memory-intensive workloads shows that under SMT colocation, ASAP is able to reduce page walk latency by an average of 25% (42% max) in native execution, and 45% (55% max) under virtualization.

CCS CONCEPTS

• **Computer systems organization → Serial architectures**; • **Software and its engineering → Virtual memory**; Allocation / deallocation strategies.

KEYWORDS

virtual memory, microarchitecture, virtualization

*This work was done while the author was at EPFL.

MICRO-52, October 12–16, 2019, Columbus, OH, USA

ACM Reference Format:
Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. 2019. Prefetched Address Translation. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3352460.3358294

1 INTRODUCTION

Massive in-memory datasets are a staple feature of many server applications, including databases, key-value stores, and data analytics frameworks. The large – and rapidly growing – data footprints, coupled with irregular access patterns, in many of these workloads result in frequent TLB misses that require a walk of the operating system's radix tree-based page table. During the walk, the levels of the radix tree-based page table (referred to as just page table, or PT) must be traversed one by one, incurring high latency overhead due to serialized accesses to the memory hierarchy.

Modern processors include several hardware features to accelerate page table walks, including hardware walkers, multi-level TLBs and translation caches. Despite these features, recent studies show that up to 50% of the performance in big-data server workloads can be lost to address translation [1]. The performance cost of address translation is destined to increase in the future due to ever-growing data working sets and larger memory capacities enabled by emerging memory technologies (e.g., Intel's 3D XPoint [2]). Eventually, these will necessitate the addition of yet another (fifth) level to the radix page table that must be visited on each page table walk. Indeed, the industry has already started preparing for the eventual transition to five-level page tables [3].

Recent research proposals seeking to ameliorate the high cost of address translation tend to fall into one of two categories: incremental improvements to the existing virtual memory subsystem and disruptive changes to it. Incremental approaches include aggressive coalescing of Page Table Entries (PTEs) within TLBs [4–6] and support for variable page sizes [7–11]. These techniques are fundamentally limited by coalescing opportunities exposed by the application and OS, as well as the capacity of the physical TLB structures.

The disruptive proposals include the use of segment-based virtual memory [12–14] and application-specific address translation [15]. While attractive from a performance perspective, these proposals require a radical re-engineering of the virtual memory subsystem at both OS and hardware levels, which presents a difficult path to adoption.

The challenge for future virtual memory systems is to enable high-performance address translation for terabyte-scale datasets without disrupting existing system stacks. As a step in that direction, this work introduces *Address Translation with Prefetching (ASAP)* – a new paradigm for reducing the latency of the iterative pointer chase inherent in PT walks through a direct access to a given level of the page table. With ASAP, a TLB miss typically exposes the

latency of just a single access to the memory hierarchy regardless of the depth of the page table.

To introduce ASAP as a minimally-invasive addition to the system stack, we exploit the observation that applications tend to have their virtual memory footprint distributed among only a handful of contiguous virtual address ranges, referred to as Virtual Memory Areas (VMAs). Each allocated virtual page has a PTE, sitting at the leaf level of the page table, and a set of intermediate page table nodes that form a pointer chain from the root of the page table to that PTE. Due to lazy memory allocation, PTEs associated with a given VMA tend to be scattered in machine memory, with no correlation between their physical addresses and that of the associated virtual pages. Our insight is that if the PTEs were to reside in contiguous physical memory and follow the same relative order as the virtual pages that map to them, then there would exist a direct mapping between the virtual page numbers in a VMA and the physical addresses of their corresponding PTEs. Given such a mapping, finding a PTE can be done through simple base-plus-offset addressing into the PTE array, avoiding the need to access preceding levels to find the PTE location. Similar logic applies to intermediate levels of the radix tree, which also can be directly indexed using base-plus-offset addressing provided the entries are in contiguous memory and in sorted order with respect to virtual addresses they map[1].

Realizing this idea requires few changes to an existing system stack. Indeed, VMAs are already explicitly maintained by modern operating systems. The key missing OS functionality is ensuring that the level(s) of the radix tree that are prefetch targets are allocated in contiguous physical memory and the entries are in sorted order. One way to achieve this is by directing the OS memory allocator to reserve, at VMA creation time, a contiguous region of physical memory for the page table entries. The required memory amounts to under 200MB for an application dataset of 100GB. On the hardware side, a set of architecturally-exposed *range registers* are needed to encode the boundaries of prefetchable VMAs. Any virtual address that misses in the TLB is checked against the range registers; on a hit, the target physical address is computed using a base-plus-offset arithmetic, and a ASAP prefetch is issued for the computed address. Multiple prefetches, to different levels of the radix tree, can be launched in parallel.

Crucially, regardless of whether a prefetch is generated or not, the PT radix tree is walked as usual. The full traversal of the radix tree guarantees that only correct prefetched entries are consumed, which helps minimize the risk of introducing a new security vulnerability into an existing architecture.

While ASAP falls short of completely eliminating page walk latencies, since it exposes typically one access to the memory hierarchy (other accesses can be overlapped, hence hiding their latency), it has one major advantage: its minimally-invasive nature with respect to the existing address translation machinery. Thus, with ASAP, TLB misses trigger normal PT walks, which are accelerated thanks to ASAP prefetches. Meanwhile, TLBs, hardware page table walkers, the PT itself and the nested address translation mechanism used for virtualization require absolutely no modifications in the presence of ASAP. Thus, ASAP can be seamlessly and gradually introduced into existing systems with no disruption to either the hardware or software ecosystem.

[1] Sorted order means that if a virtual page number X comes before virtual page number Y, then the radix tree entry for X resides at a physical address less than that of the radix tree entry for Y.

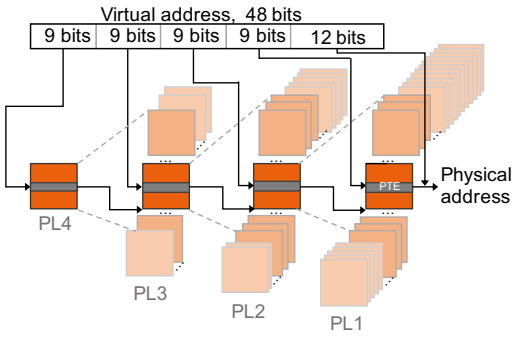


Figure 1: Linux/x86 Page Table as four-level radix tree.

To summarize, this paper's contributions are as follows:

- We quantitatively show that ever-growing datasets, colocated workloads and virtualized deployments all pressure existing address translation mechanisms. For example for `memcached`, by increasing the dataset size by 5×, average page walk latency grows by 1.2×. SMT colocation and virtualization increases page walk overhead by 2.7× and 5.3×, respectively.
- We introduce Address Translation with Prefetching, which affords direct access to target entries of the PT radix tree, thus hiding the latency of the pointer-chasing PT walk. ASAP works in concert with unmodified existing address translation mechanisms, is fully compatible with virtualization, does not allow speculative usage of prefetched entries and requires minimally-invasive modifications to the system stack.
- We demonstrate that ASAP is able to reduce average page walk latency by 14% (20% max) and 39% (43% max) on average when executed in native and virtualized environments, respectively. Under SMT colocation, ASAP's ability to shorten average page walk latency increases to 25% average (42% max) for native and 45% average (55% max) for virtualized environments.

2 MOTIVATION

2.1 Virtual Memory Basics

A process' page table (PT) maintains the information about how the process' virtual space is mapped onto physical memory. Each PT entry (PTE) contains a per-page virtual-to-physical mapping as well as auxiliary metadata, such as access permissions bits. Upon each memory access, the CPU must find the corresponding PTE, determine the address translation and validate the access type against the permission bits.

On x86, the PT is organized as a four-level radix tree as shown in Figure 1. The leaves of the radix tree, i.e., PT Level 1 (PL1), contain PTEs. A given PTE contains a translation for all virtual addresses that belong to a single OS page, as well as permission and other bits that the OS uses. While the primary role of the PT is address translation, the OS extensively uses PTs for other essential memory management mechanisms, such as copy-on-write, accessed and

Table 1: Increase in memcached page walk latency under various scenarios. The data is normalized to native execution in isolation with a 80GB dataset.

5× larger dataset	SMT colocation	Virtualization	Virtualization + SMT colocation
1.2×	2.7×	5.3×	12.0×

dirty pages management and the I/O associated with memory-mapped files. The general-purpose organization of the PT radix tree allows to flexibly track both sparse and dense memory regions, allowing reasonable look-up time and providing an efficient way to monitor and analyze memory regions' usage and evolution. To minimize the number of levels in the tree, each intermediate node of the tree has high fan-out having up to 512 child nodes that are indexed with the corresponding subset of the address bits, so that each PT level can be traversed with a single memory look-up.

Modern CPUs provide an ensemble of mechanisms to enable fast address translations. Translation Lookaside Buffer (TLB) is a hardware-managed structure that caches the virtual-to-physical translations to frequently accessed data. When the TLB does not contain the required translation (i.e., a TLB miss occurs), a hardware state machine, called a page walker unit, brings the missing PT entry from memory and installs the translation in the TLB, or raises a page fault if the appropriate PTE is not found. Upon a TLB miss, the page walker traverses the PT radix tree level-by-level chasing up to four pointers in physical memory. The process of traversing the PT radix tree, called *a page (table) walk*, can take hundreds of CPU cycles depending on the footprint and locality characteristics of the PT [16, 17]. To speed up the page walks, processors rely on conventional cache hierarchy to naturally cache recently accessed PT entries.

In addition to leveraging the conventional cache hierarchy, CPUs feature dedicated *Page Walk Caches (PWCs)*. PWCs contain frequently traversed intermediate nodes of the radix tree, allowing the page walker to bypass one or few levels of the PT on PWC hits. However, due to silicon area and power constraints, PWCs are highly limited in size, featuring just a few tens of entries [18].

To support virtualization, most commercial CPUs implement nested page tables where host and guest OSes manage their own sets of page tables [19]. When a process running in a guest OS experiences a TLB miss, it has to perform a 2D walk of the nested PT. In a 2D walk, each access to the guest PT node causes a full 1D walk (i.e., 4 accesses to the memory hierarchy) in the host PT to find the next guest PT node, and one final walk to access the data. As a result, a single page walk can cause up to 24 memory accesses, leading to much higher translation overhead even after extending page walk caches for both dimensions of the 2D walk [20].

2.2 Performance Cost of Address Translation

To estimate translation overheads on modern hardware, we evaluate a number of popular benchmarks and big memory applications running in isolation and in colocation with a memory-intensive co-runner (see §4 for details of the methodology). Figure 2 shows that for native applications, page walks account for up to 82% of CPU cycles due to large memory footprints and, depending on the application, irregular memory access patterns. This combination tends

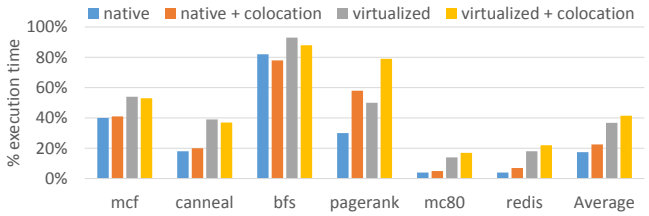


Figure 2: Fraction of execution time spent in page walks. mc80 is memcached with an 80GB dataset.

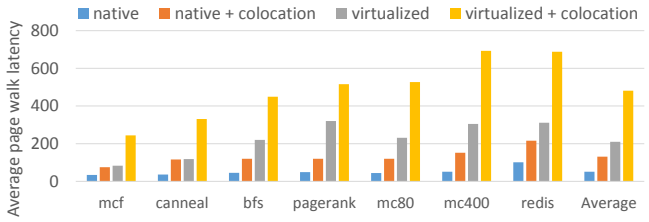


Figure 3: Average page walk latency in various scenarios.

to result in poor spatio-temporal locality in the address stream, which all existing mechanisms for accelerating address translation (TLBs, PWCs, processor's cache hierarchy) exploit to get good performance. Under virtualization, the overhead of address translation can be as high as 93% of CPU cycles due to the added cost of the nested page walk.

Figure 3 demonstrates that, despite all existing hardware support, page walk latency may climb to hundreds of cycles. Table 1 summarizes the trend by focusing on the memcached workload. Various factors can affect the page walk latency, including the size of the dataset, interference caused by a co-running application and virtualization. If running on a core in isolation and in native mode, memcached has average page walk latency of 44 cycles. Increasing the dataset size of memcached from 80GB to 400GB causes the average page walk latency of memcached to climb by 1.2×. Colocation with another application on the same core (see Section 4 for methodology details) increases the page walk latency by 2.7× (to 120 cycles). Virtualization (without colocation) increases the page walk latency of memcached by 5.3×. Finally, virtualization with colocation propels the page walk latency of memcached to 527 cycles, a whopping 12× increase.

2.3 Can Large Pages Help?

One of the least intrusive, with respect to the PT radix tree, innovations in memory management has been the introduction of large pages. This approach allows to replace 512 contiguous small pages with a large page of the equivalent size. The hierarchical structure of the PT radix tree naturally supports large pages, requiring modest hardware and software extensions. For example, in Linux/x86, a large page of 2MB is managed by a single entry in the PL2 level of the PT, replacing the corresponding 512 independent PT entries in the PL1 level.

Unfortunately, even the least intrusive changes in the virtual memory mechanisms carry numerous implications on the overall system behavior and introduce performance pathologies. On x86,

the introduction of 2MB and 1GB large pages revealed an ensemble of unexpected problems. Araujo et al., for example, show that the use of large pages leads to memory fragmentation in multi-tenant cloud environments [21]. Under high memory fragmentation, Linux often has to synchronously compact the memory before a memory chunk of the necessary size can be allocated, introducing high average and tail latencies. Kwon et al. showcase a problem in unfair large pages distribution among multiple applications sharing a single server, as well as point to increasing memory footprint due to the internal fragmentation [8]. For instance, the authors show that `redis` increases its memory footprint by 50% when using large pages and may start swapping even with a carefully provisioned physical memory.

While being a simple and natural idea, the systems community has struggled with wide adoption of large pages. The root of the problem is the lack of memory management flexibility (e.g., copy-on-write in OS, deduplication and ballooning optimizations in hypervisor [9, 22]) with large pages as compared to fine-grain paging.

2.4 Disruptive Proposals are Undesirable

With rapidly increasing application dataset sizes and the associated growth in translation overheads [23], some of the recent work argues for a complete replacement of the PT radix tree and the address translation mechanisms that are designed around it. One set of proposals has suggested using large contiguous segments of memory instead of fine-grained paging for big-data server applications [12–14]. These proposals are based on observations that server applications tend to allocate all of their memory at start-up and make little use of page-based virtual memory mechanisms, such as swapping and copy-on-write [12–14].

While tempting from a performance perspective, there are several major issues with the segment-based approach that prevent its adoption. First, segment-based memory relies on specific characteristics of a single class of applications, whereas an OS has to be general-purpose. Hence, the OS needs to efficiently support both the conventional page-based approach and the segment-based one. Supporting two separate translation mechanisms makes the memory subsystem heterogeneous and more complicated to manage. Second, not all big-memory applications can operate on a small number of segments, as noted by [6], while supporting a large number of segments is impractical in hardware, as noted in the original paper [14].

Another challenge with segment-based memory management is that, in practice, finding large contiguous physical memory regions can be challenging in the presence of memory cell failures. Such failures are exposed to the OS, which monitors the health of the available physical memory, detects the faulty cells, and *retires* the affected physical memory at granularity of small pages [24, 25]. While a recent study at Facebook already reports increasing memory error rates due to DRAM technology scaling to smaller feature sizes [26], emerging memory technologies, such as 3D XPoint, are likely to have a higher incidence of memory cell failures due to lower endurance as compared to DRAM [27, 28]. Hard memory faults dramatically complicate memory management not only with segments but also with large pages; e.g., by introducing an extra level of abstraction [29] and new software and hardware machinery (e.g., per-segment Bloom filters for pages with hard faults [13]).

Other prior work investigated application-specific address translation, allowing the developers to choose address translation methods appropriate for their applications [15]. However, these approaches expose the complexity of virtual memory programming to the application developers or, if implemented at the system level, lead to an increase in OS memory management complexity. Both greatly impede the adoption of these ideas in the wild.

2.5 Incremental Approaches are Insufficient

To avoid the disruption to existing system stacks, some researchers have proposed microarchitectural and OS-based techniques to accelerate address translation while retaining compatibility with the conventional PT and radix tree mechanisms. One such group of mechanisms seeks to leverage existing contiguity in both virtual and physical space to coalesce multiple PTEs into a single TLB entry so as to increase TLB reach [4, 5].

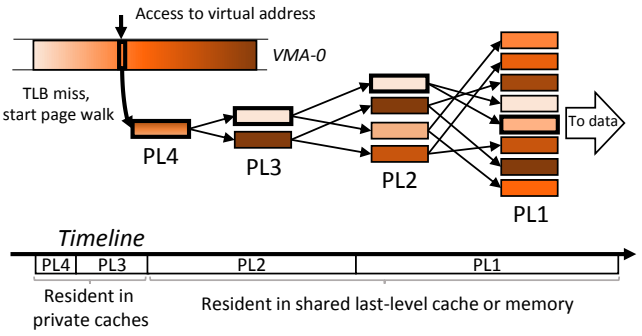
Problematically, the effectiveness of coalescing-based approaches is fundamentally constrained by the size of a TLB structure. Another limitation is that coalescing relies on having contiguity in the physical memory space of an application; however, ensuring such contiguity is not an objective of existing memory allocators, such as the Linux buddy allocator [6]. As a result, application contiguity characteristics can vary greatly across different runs of the same application, making solutions that rely on PTE coalescing unreliable from a performance perspective.

Another set of techniques focus on improving TLB efficiency in the presence of multiple page sizes. A straight-forward TLB design that statically partitions capacity across a set of supported page sizes will suffer from poor utilization when one page size dominates. Moreover, because the size of the page that belongs to a particular memory access is unknown before a TLB look-up, all of the TLB structures need to be checked, increasing TLB hit latency and energy consumption. While recent works have attacked these problems [7, 10, 11], their effectiveness is fundamentally limited by the capacity of the TLB structure.

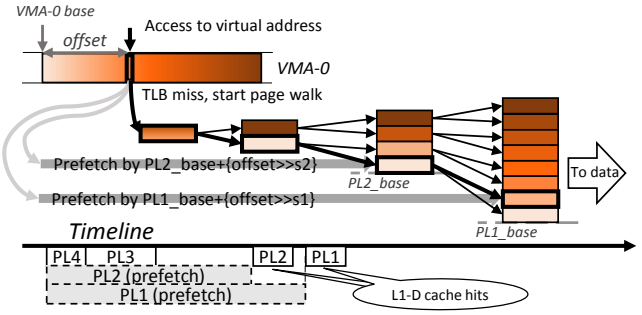
2.6 Virtual Memory Tomorrow

Devised decades ago, the page table and radix tree-based indexing mechanism have formed the basis for all of modern address translation machinery. The fact that these mechanisms have survived for so long, and have been extended – rather than replaced – to support virtualization, speaks to their flexibility and the importance of backward compatibility.

With the unrelenting growth in application dataset sizes [23] and the advent of high-capacity storage-class memories [2, 30, 31], future systems will have to contend with much higher pressure on address translation structures, including TLBs and caches. Moreover, to accommodate large memory footprints, the industry has started preparing to add an additional, fifth, level to the page table [3]. The extra level will increase the footprint of the radix tree, making it more challenging to capture in processor's caches, and will also increase the depth, and hence the latency, of the page walk. The time is ripe for new ideas on accelerating address translation without disrupting existing software stacks.



(a) Conventional page walk and its timeline.



(b) ASAP: Page walk accelerated with prefetching PT nodes from the bottom PT levels, pre-sorted by virtual address.

Figure 4: Conventional and ASAP-accelerated page table walks. Darker (lighter) PT node colors are associated with higher (lower) virtual addresses of the corresponding data pages.

3 PREFETCHED ADDRESS TRANSLATION

3.1 Overview

In this work, we aim to lower page walk costs in a non-disruptive manner, retaining full compatibility with the radix tree-based page table and the existing address translation machinery (TLB, page walk caches, etc). Because the page walk time is dominated by a pointer chase through the levels of the page table (Figure 4a), we introduce Address Translation with Prefetching – a mechanism to prefetch page table nodes ahead of the page walker. We focus on the deeper levels of the PT (PL1 and PL2) as the most valuable prefetch targets, because the fourth (PL4) and third (PL3) PT levels are small and efficiently covered by the Page Walk Caches and the regular cache hierarchy. Meanwhile, the second (PL2) and the first (PL1) levels are much larger and often beyond reach for on-chip caching structures for big datasets. For instance, for a 100GB dataset, the footprint of the PT levels is 8B, 800B, 400KB and 200MB for PL4, PL3, PL2 and PL1, respectively.

To reduce the page walk latency, we propose issuing prefetches for the PL1 and PL2 levels concurrently with the page walker initiating its first access (i.e., to PL4, which is the root of the PT), as demonstrated in Figure 4b. Triggered on a TLB miss, the prefetcher needs to determine the physical addresses of the target PT nodes in both PL1 and PL2 levels of the page table. This is accomplished via a simple base-plus-offset computation, enabled through an ordering of memory pages occupied by the page table as discussed below. Prefetches travel like normal memory requests through the memory hierarchy and are placed into the L1-D, thus maximally repurposing the existing machinery.

Critically, with ASAP, the page walker performs the full walk as usual, consuming the prefetched entries. By executing the page walk, ASAP guarantees that only correct entries are consumed, which enables proper handling of page faults and reduces the risk of introducing a new security vulnerability into an existing architecture.

To introduce ASAP as a natural addition to the existing system stack, we exploit the observation that a process operates on few contiguous virtual address ranges. One important example of such a range is the heap, which forms a large contiguous region in the virtual space of a process. Each allocated virtual page inside a virtual address range has a PTE at the leaf level of the PT, reached through a chain of PT nodes, one per PT level (Figure 4a). Thus, there exists a one-to-one mapping between a virtual memory page and a corresponding PT node at each level of the PT radix tree. However, this correspondence exists only in the virtual space, but not in the physical space, due to the buddy allocator that scatters the virtual pages, including those of the PT, across physical memory.

To enable ASAP, there needs to be a direct mapping from a virtual page to a corresponding PT node in physical memory (shown by the grey Prefetch arrows in Figure 4b). Our insight is that if the PT nodes for a given level of the page table in physical memory follow the same order as the virtual pages they map to, then a direct index into the PT array is possible using simple base-plus-offset computation. The solution is to have the OS induce the required ordering for the PT nodes in physical memory. As discussed below, this requires straight-forward extensions in the kernel and absolutely no modifications to the actual page table structure.

In the remainder of the section, we discuss key aspects of ASAP including existing contiguity in the virtual address space, how contiguity can be induced in the radix tree-based page table, architectural support for ASAP, and virtualization extensions.

3.2 Virtual Address Contiguity

Virtual addresses in page tables appear as a set of contiguous virtual ranges that are defined by the way the applications create and use virtual address spaces, such as heap and stack. In Linux, the OS manages these ranges using a virtual memory area (VMA) tree that contains the information about all non-overlapping virtual address ranges (further referred to as VMAs) allocated to a process. Other OSes have data structures analogous to Linux VMA tree, e.g., Virtual Address Descriptor (VAD) tree in Windows [32].

The applications we studied allocate few VMAs that stay stable during their execution. Our results (Table 2) show that a small number of VMAs cover 99% of the application footprint. These few large VMAs are attributed to heap and memory-mapped regions that contain the application data structures that are the primary causes of page walks. Meanwhile, small VMA mostly represent dynamically-linked libraries and the stack, which are frequently accessed and rarely cause TLB misses due to high temporal reuse.

Table 2: Total number of VMAs, number of VMAs that cover 99% of footprint, number of contiguous regions in physical memory, and total number of PT pages per application.

Application	Total VMAs	VMAs for 99% footprint coverage	Contig. phys. regions	PT page count
canneal	18	4	487	2842
mcf	16	1	626	3189
pagerank	18	1	2076	38504
bfs	14	1	4285	66015
mc80	26	6	1976	45878
mc400	33	13	5376	213097
redis	7	1	3555	44171

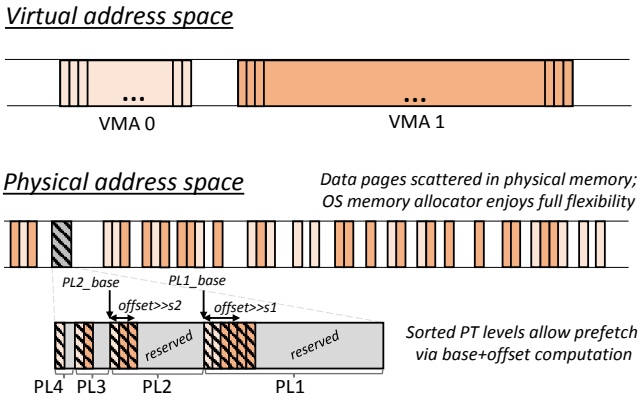


Figure 5: Virtual and physical memory layout with ASAP. The pages that contain the PT are color-coded according to their corresponding VMAs.

3.3 Inducing Contiguity in the Page Table

As explained in the previous section, the virtual address space enjoys high contiguity. However, when it comes to physical memory, the pages of the PT are often scattered. For the applications we studied, the number of contiguous physical memory regions that store the PT nodes can reach into thousands (Table 2). The reason for such lack of locality in the page table is that PT nodes, just as any other data in Linux, are lazily allocated in pages whose position in physical memory is determined by the Linux buddy allocator. The buddy allocator optimizes for allocation speed, allocating pages on demand in first available slots in physical memory. The result is a complete lack of correspondence between the order of virtual pages within a VMA and the physical pages containing PT nodes.

To enable ASAP, the OS needs to guarantee that PT nodes within each PT level are located in contiguous physical memory according to the their corresponding virtual page numbers within a VMA of the process, conceptually shown in Figure 4b. There are two ways to achieve such placement of PT nodes in physical memory: first is to deploy a custom allocator to enforce page ordering and contiguity in physical memory for the PT, and second is to sort the already allocated PT nodes in the background. While both approaches are plausible, we focus on the former as a concrete case study.

In Linux, a VMA tree contains all ranges of virtual addresses (further referred to as VMAs) that the OS provides to the process

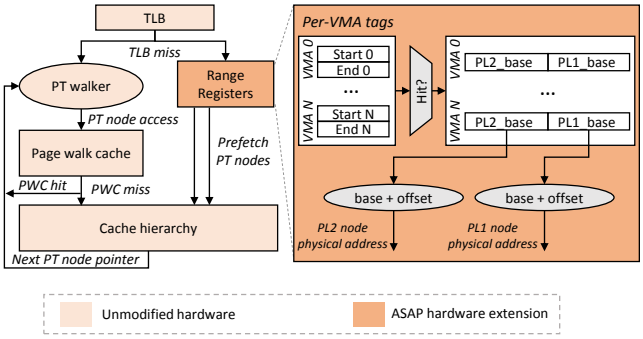


Figure 6: Architectural support for ASAP.

per its request. According to the demand paging and lazy allocation principle, which is employed in most operating systems, including Linux, the VMAs are created immediately, e.g., upon an `mmap` system call, whereas PT nodes are created and populated only upon a first access to the corresponding virtual addresses, which cause page faults that lead to creation of the corresponding virtual-to-physical mappings in the PT. Hence, each VMA defines how many mappings will eventually appear for it in the PT, and what portion of the page table the VMA will occupy in physical memory.

Since the OS knows the beginning of each VMA in the virtual space and its size, the OS can reserve contiguous physical memory regions for PT nodes at each level of the page table ahead of the eventual demand allocation of page table entries. When these are (lazily) populated, the OS can further enforce the ordering of PT nodes within each PT level to guarantee that it matches the ordering of the virtual pages mapping to them. Doing so ensures both *contiguity* and *ordering* of PT pages in machine memory, which enables direct indexing into a given level of the page table.

Figure 5 shows the layout of virtual and physical address spaces with ASAP. The virtual space layout and the layout of data pages in physical memory remains the same as in vanilla Linux; the only change required by ASAP is the introduction of contiguous physical memory regions for pages containing PT nodes.

Cost. Unlike prior work on direct segment addressing [12–14], which requires allocating the entire dataset of an application within large contiguous physical regions (see Section 2.4 for a discussion of the drawbacks), ASAP requires contiguity in only a tiny portion of the physical memory thanks to the compact nature of the PT radix tree. For example, for an application with a 100GB dataset, PL4 and PL3 levels together fit in a single 4KB page, PL2 requires 400KB, and the leaf PL1 level necessitates around 200MB. This example shows that the physical memory footprint that must be guaranteed contiguous by the OS to hold the sorted PT nodes amounts to a mere 0.2% of an application's dataset size.

3.4 Architectural Support

Figure 6 shows the microarchitecture of ASAP. As the figure shows, ASAP non-disruptively extends the TLB miss-handling logic. For each VMA that is a prefetch target, ASAP requires a VMA descriptor consisting of architecturally-exposed *range registers* that contain the start and end addresses of the VMA, as well as the base physical

addresses of the contiguous regions containing the 1st (PL1) and 2nd (PL2) PT levels mapping the VMA. ASAP's VMA descriptors are part of the architectural state of the hardware thread and are managed by the OS in the presence of the events like a context switch or interrupt handling. According to the results from §3.2, tracking 8–16 VMAs is enough to cover 99% of the memory footprint for the studied benchmarks.

With ASAP, each TLB miss triggers a lookup into the range registers, which happens in parallel with the activation of the page walker. The lookup checks the virtual address of the memory operation against the tracked VMA ranges; on a hit, target prefetch addresses in PL1 and PL2 are calculated with a base-plus-offset computation using each level's respective base physical addresses and the offset bits from the triggering virtual address. Note that the actual offset differs between PL1 and PL2, and is derived for each of these PT levels by simply shifting the incoming offset bits by a fixed amount (labeled `s1` and `s2` in Figure 6). The prefetch requests to the two target PT nodes are then issued to L1-D if it has a port available. As a result, the cache lines containing PT nodes are loaded into the L1-D, from which they will be subsequently accessed by the page walker.

An important aspect of ASAP is that it requires no modifications either to the cache hierarchy or to the page walker. ASAP leverages existing machinery for buffering the outstanding prefetch requests in L1-D's MSHRs and buffering the data brought in by ASAP in the L1-D itself. Prefetches are thus best-effort (e.g., not issued if an MSHR is not available). In contrast to data prefetchers, ASAP does not noticeably increase memory bandwidth pressure since ASAP prefetches are nearly always correct (except in the special cases of "holes" in a PT range, as discussed in §3.7.2), effectively converting the page walker's demand misses into prefetches.

3.5 ASAP for Large Pages and Five-Level PT

Thanks to the recursive structure of the PT radix tree, no modifications are required to support large pages of any size. Translations that correspond to large pages are stored one or two levels above from the leaf (PL1) PT level. For example, page table entries for 2MB pages are stored in the 2nd (PL2) level of the PT radix tree. Since the size of the page is unknown before the page walker inspects the ultimate PT node (e.g., the PT node at PL2 contains a dedicated bit that distinguishes a 2MB page PTE from a pointer to the PL1 node that contains a 4KB page entry), some of the prefetch requests may be redundant (e.g., a request to the PL1 node if 2MB pages are used).

With the advent of five-level page tables, ASAP can be naturally extended to issue an additional prefetch request to the added PT level.

3.6 ASAP for Nested Walks

In virtualized environment, ASAP can be applied in both guest and host dimensions, which presents a significant acceleration opportunity due to the high latency of nested page walks. Under virtualization, the radix tree levels of both guest PT (gPT) and host PT (hPT) targeted by ASAP must be contiguous and ordered in the host physical memory. Similar to the native setup, this must be ensured by the hypervisor and guest OS.

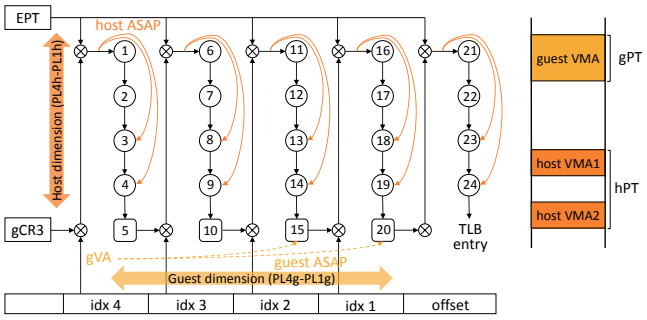


Figure 7: Nested prefetched address translation. Accesses are enumerated according to their order in a 2D page walk.

In the general case, 2D walk starts by reading the gCR3 register that stores the location of the gPT root, followed by consecutive 1D walks in the host to access each of the gPT entries. Figure 7 shows the 2D walk with ASAP prefetching, assuming ASAP is configured to prefetch the 2nd (PL2) and 1st (PL1) levels of gPT and hPT. Immediately at the 2D walk start, ASAP issues prefetch requests to the gPT nodes in the PL2 and the PL1 levels to overlap accesses 15 and 20 with the previous ones. Then, just as the page walker starts the 1D walk in the host (steps 1–4), ASAP issues prefetch requests to the PL2 and PL1 levels of the hPT using the guest physical address of the gPT root. The process repeats for each 1D walk in host, namely steps 6–9, 11–14, 16–19, 21–24.

From the software perspective, to enable ASAP-accelerated 2D walks, the guest OS requires minimal modifications. Similar to the native case, the guest OS needs to ensure contiguity in the physical memory regions storing PL1 and PL2 levels of the page table. Under virtualization, the guest must make these requests to the hypervisor, and notify the hypervisor when any of these regions need to be extended. Thus, on x86, the guest OS' system calls that change the contents of the (guest) VMA tree must execute `vmcall` instructions to trigger the transition into the hypervisor so that it can invoke the host OS' PT allocator to guarantee the region's contiguity in both host and guest physical spaces.

From the hardware perspective, accelerating address translation in the host with ASAP requires additional range registers. Crucially, we observe that in Linux/KVM virtualization, from the perspective of the host OS, an entire guest VM is a process that has a *single* virtual address region [19]. Hence, a single set of range registers is sufficient to map the guest VM (including the target page table) as a host VMA, allowing acceleration of walks in the host dimension (e.g., steps 1–4 or 21–24 in Figure 7). Meanwhile, the number of VMAs in the guest OS is unaffected by virtualization, requiring the same number of range registers for ASAP acceleration as in the native environment.

3.7 Discussion

3.7.1 Page Fault Handling. Since most OS'es follow the lazy allocation principle, the PT is populated with mappings on demand, i.e., the first access to a non-allocated page causes a page fault, leading to the mapping being created in the PT. Thus, both with and without ASAP, some of the PT nodes corresponding to a VMA region will stay uninitialized until the first access happens.

Table 3: Workloads used for evaluation.

Name	Description
mcf	SPEC'06 benchmark (ref input)
canneal	PARSEC 3.0 benchmark (native input set)
bfs	Breadth-first search, 60GB dataset (scaled from Twitter)
pagerank	PageRank, 60GB dataset (scaled from Twitter)
mc	Memcached, in memory key-value cache, 80GB and 400GB datasets
redis	In-memory key-value store (50GB YCSB dataset)

In this presence of ASAP, this behavior does not impact correct page fault handling. Thus, when the page walker performs a page walk that eventually triggers a page fault, ASAP still issues prefetch requests to PT nodes in PL1 and PL2. These prefetches accelerate page fault detection by the hardware walker.

3.7.2 VMAs Evolution. Most VMAs in the VMA tree belong to well-defined process segments, such as heap, stack, memory-mapped files and dynamic libraries. The largest data segments – the ones that hold the application dataset, such as heap and the memory-mapped segments – can grow or shrink in a pre-determined direction as the process continues its execution. For instance, upon a `malloc` call, the allocator may grow the heap segment by invoking `brk/sbrk` system calls to extend the segment towards higher virtual addresses.

To extend the contiguous reserved PT regions in the event of a VMA extension, the OS needs to request memory from the buddy allocator next to the boundary of the existing region. Unfortunately, the buddy allocator does not optimize for contiguous region allocations, usually providing the first best fit chunk of physical memory. Furthermore, the physical memory next to the border of the region can be already allocated (e.g., for regular data pages). To avoid changing the buddy allocator mechanisms, we argue for asynchronous regions extension in the background[2], triggered by a system call that extends the corresponding VMA. Thanks to lazy PT population with newly allocated mappings, the OS has time to clear the adjacent memory area, moving already allocated pages elsewhere in physical memory.

In the unlikely event that the OS cannot free some of the pages in the region extension area (e.g., if the pages are pinned), it can allocate some of the PT pages apart from the reserved region in the VMA. Thanks to the pointer-based structure of the PT radix tree, the page walker will be able to correctly walk the page table as usual. The only consequence of such "holes" in the reserved PT regions is that the page walks that target the PT entries located in the "holes" would not be accelerated by ASAP.

4 METHODOLOGY

To evaluate ASAP, we employ a methodology similar to prior work in this space, which reports TLB-miss induced overheads and page walk latencies [12, 14, 15, 18]. We measure TLB miss overheads on real hardware using performance counters. Given our focus on long-running big-memory workloads, we find full-system simulations intractable for projecting end-to-end performance. Instead, we report average page walk latency obtained from a detailed memory hierarchy simulator that models processor caches, page walk caches and TLBs.

[2] A similar mechanism is employed by Transparent Hugepage Support [33] daemon that can compact pages per a request/hint communicated by the application via `madvise` system call (e.g., `MADV_MERGEABLE` advice value).

Table 4: Parameters of the hardware platform used for performance counters studies and memory trace generation.

Parameter	Value/Description
Processor	Dual socket Intel(R) Xeon(R) E5-2630v4 (BDW) 2.40GHz 20cores/socket, SMT
Memory size	160GB/socket (768GB/socket for `memcached_400GB`)
Hypervisor	QEMU 2.0.0, 128GB RAM guest
Guest/host operating system	Ubuntu 14.04.5, 4.4.0-31-generic kernel

Table 5: Parameters used in simulation.

Parameter	Value/Description
L1 I/D-TLB	64 entries each, 8-way associative
L2 S-TLB	1536 entries, 6-way associative
Page walk caches	3-level Split PWC: 2 cycles, PL4 - 2 entries, fully assoc.; PL3 - 4 entries, fully assoc.; PL2 - 32 entries, 4-way assoc. (similar to Intel Core i7 [35]) Virtualization: one dedicated PWC for guest PT, one for host PT.
L1 I/D cache	32KB, 8-way associative, 4 cycles
L2 cache	256KB, 8-way associative, 12 cycles
L3 cache	20MB, 20-way associative, 40 cycles
Main memory	191 cycles access latency

Our evaluation primarily focuses on small (4KB) pages, since fine-grain memory management delivers greater flexibility, better memory utilization and better performance predictability (Section 2.3). We explore the effect of large pages in Section 5.2.

Benchmarks. We select a set of 6 diverse applications that exhibit significant TLB pressure (6-85% L2 TLB miss ratio) from SPEC'06, PARSEC 3.0, graph analytics (atop of Galois framework [34]) and in-memory key-value stores. For the graph applications (`bfs` and `pagerank`), we used a 60GB synthetic dataset with edge distribution modeled after a (smaller) publically-available Twitter dataset. The applications and datasets are listed in Table 3.

Measuring the overhead of TLB misses. We measure the fraction of execution time when the page walker was active. This is done on a real system, whose parameters are listed in Table 4. Using the Linux `perf` tool, we collect the values of two performance counters reporting (1) cycles when page walker is active and (2) total number of execution cycles. We calculate TLB miss overhead as the number of cycles when page walker is active to total execution cycles.

Measuring page walk latency. As a primary evaluation metric for ASAP, we use page walk latency. We functionally model memory hierarchy of an Intel Broadwell-like processor using a simulator based on DynamoRIO [36]. The parameters of the simulated CPU are shown in Table 5.

On every TLB miss, we simulate a page walk using application's page table dump, captured through an in-house kernel module. For

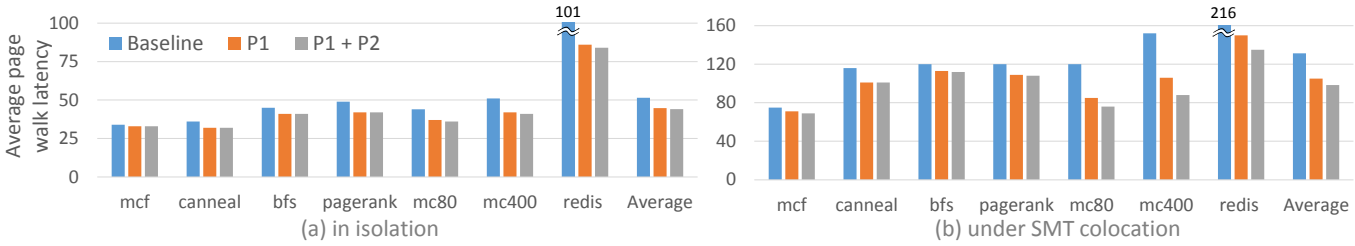


Figure 8: Average page walk latency in native execution (a) in isolation, (b) under SMT colocation. Lower is better.

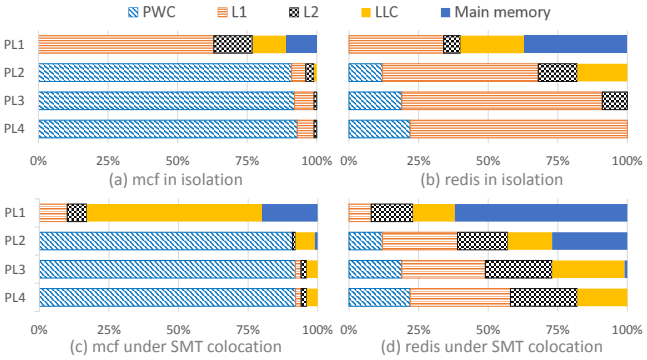


Figure 9: Fraction of page walk requests served by each level of the memory hierarchy for a given PT level.

each access to the memory hierarchy during a page walk, we trace and record the levels of the memory hierarchy involved in serving the access. Since a page walk is a serial pointer chasing operation, we calculate the page walk latency by adding up access latencies of all memory hierarchy levels involved in each page walk trace.

Workload colocation. In modern datacenter and cloud environments, applications are aggressively colocated for better CPU and memory utilization [37]. Indeed, Google reports that they aggressively colocate different applications on SMT cores as a routine practice [38].

We simulate a colocation scenario on a dual-threaded SMT core by placing a memory-intensive co-runner thread alongside the studied application thread. To avoid biasing the study by the idiosyncrasies of any one application, we use a synthetic co-runner that issues one request to a random address for each memory access by the application thread. Colocation pressures the cache hierarchy, which is used to cache page table entries (from both intermediate and leaf nodes of the page table), hence increasing the average walk duration.

Note that we do not model contention in the TLBs and PWCs stemming from colocation. Contention in these structures would result in more/longer page walks, thus increasing the opportunity for ASAP. Thus, our speed-up estimates for ASAP under colocation are conservative.

Virtualization. To assess ASAP in a virtualized environment, we record the guest PT contents using an in-house kernel module. On the host side, we model the layout of the PT in a system without ASAP by mimicking the Linux buddy allocator's behavior by randomly scattering the PT pages across the host physical memory. To model ASAP, we maintain PL1 and PL2 pages in contiguous regions in the host.

5 EVALUATION

5.1 ASAP in Native Environment

We first evaluate ASAP under native execution, first without then with colocation. We evaluate several configurations. The first is the baseline, which does not employ ASAP and corresponds to a design representative of existing processors. We study two ASAP configurations: the first of these (referred to as *P1*) prefetches only from PL1 level of the page table; the second (referred to as *P1+P2*) prefetches from both PL1 and PL2 levels.

5.1.1 ASAP in Isolation. Figure 8a shows the average page walk latency for the baseline and both ASAP configurations when the application executes without a corunner. In the baseline, page walk latency varies from 34 to 101 cycles, with an average of 51 cycles. The largest latency is experienced by `redis`.

Prefetching only PL1 reduces average page walk latency by 12% over the baseline (to 45 cycles). In absolute terms, the largest observed latency reduction is on `redis`, whose page walk latency drops by 15 cycles. In contrast, `mcf` experiences only 1 cycle reduction in average page walk latency. The difference in efficacy of ASAP for these applications can be explained by understanding from which level of the memory hierarchy page walk requests are served.

Figure 9 shows the fraction of requests satisfied by a given level of the memory hierarchy for each level of the page table traversed in a walk. In the case of `mcf` running in isolation (Figure 9a), requests to all levels except PL1 mostly hit in PWC and are satisfied within a few cycles; meanwhile, requests to PL1 take considerably longer, with nearly a third of requests hitting in L2, LLC or main memory. Because the page walker traverses PL1 through PL3 so quickly thanks to PWC hit, ASAP has little opportunity to hide latency on this workload. Meanwhile, `redis` hits in PWC much less frequently as shown in Figure 9b; in particular, a significant fraction of page walker's requests to PL2 reaches the L2 or LLC, which provides ASAP with an opportunity to overlap its prefetch to PL1 with the page walk to previous levels.

Despite the fact that a considerable number of page walks can hit in PWC, increasing PWC capacity does not substantially reduce page walk latency. We observe that when running in isolation, doubling the capacity of each PWC with respect to the default

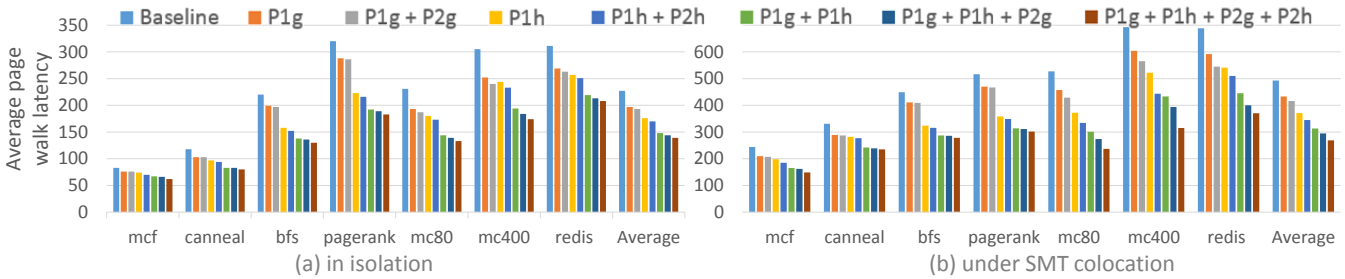


Figure 10: Average page walk latency with virtualization (a) in isolation, (b) under SMT colocation. Lower is better. Note the different scaling of y-axis between the subfigures.

configuration provides a negligible page walk latency reduction – 2% and 3% in native and virtualized scenarios, respectively. These results corroborate industry trends: PWC capacity has not grown beyond 32 entries per level in several recent Intel processor generations from Westmere to Skylake [39].

Prefetching from PL2 in addition to PL1 reduces average page walk latency by 14% over the baseline – a small improvement over prefetching just PL1. The reason for such limited benefit of prefetching from an additional level of the page table can be understood by examining Figures 9a and 9c). Because the vast majority of requests to PL3 and PL4 hit in PWC or the L1-D, there is little opportunity for ASAP to overlap these accesses with prefetches to PL2.

5.1.2 ASAP under Colocation. Figure 8b shows page walk latency for native execution under colocation. When the memory hierarchy experiences additional pressure due to the presence of a memory intensive co-runner, page walk latency increases as compared to execution in isolation as PT nodes are more likely to be evicted from the caches. Comparing Figure 9c to Figure 9d, one can see that under colocation, there are considerably fewer page walk requests served by the L1-D cache than when running in isolation. As a result, the average page walk latency on the baseline with colocation ranges from 74 to 216 cycles, with an average of 131 cycles. This represents an increase of 2.1-3.2× (average of 2.6×) over execution in isolation.

With prefetching only to PL1, ASAP achieves a page walk latency reduction of 20%, on average, and 31% in the best case (on `redis`, whose average page walk latency drops by 66 cycles). When page walks frequently contain more than one long-latency request – such as when requests to PL1 and PL2 are both served by the main memory, as in Figure 9d – ASAP's ability to shorten the page walks latency significantly improves by overlapping the latency of these requests.

Prefetching PL2, in addition to PL1, is also more beneficial in the presence of a co-runner. Prefetching both levels reduces the page walk latency by 25% on average and up to 42% (on `memcached` with 400GB dataset), over the baseline.

5.2 ASAP under Virtualization

To understand ASAP's efficacy in a virtualized setting, we study several ASAP configurations that prefetch from PL1 only or from both PL1 and PL2 in the guest and/or host. The baseline is a system without ASAP.

Results for execution in isolation are shown in Figure 10a. Under virtualization, the baseline page walk latency ranges from 83 to 320 cycles, with an average of 227, a 4.4× increase in comparison to native execution due to the high cost of 2D walks. We observe that prefetching from PL1 of only the guest (*P1g* in the figure) reduces the average page walk latency by 13% on average. Prefetching from both PL1 and PL2 (*P1g+P2g*) in the guest shortens the page walk latency by another 2%, totaling a 15% average reduction over the baseline. Such modest results can be explained by the fact that the nested page walk spends most of its time traversing the host page table (Section 3.6), which is not accelerated by ASAP that prefetches from only the guest page table.

When ASAP prefetches from PL1 of the guest *and* from PL1 of the host together (*P1g+P1h*), walk latency decreases by 35% on average. Not surprisingly, the highest performance is attained if both PL1 and PL2 are prefetched in both guest and host (*P1g+P1h+P2g+P2h*). In that case, page walk latency decreases by 39% on average, and 43% (on `pagerank`) in the best case. In absolute terms, this configuration reduces page walk cycles by 88 on average and 137 max (on `pagerank`).

In a virtualized setting with workload colocation, there exists a larger opportunity for ASAP to capitalize on. Figure 10b shows that the baseline page walk latency under colocation increases considerably (on average, 493 cycles with colocation versus 227 cycles without), which indicates that there are more long memory accesses which can be overlapped with ASAP. Prefetching from PL1 in both guest and host reduces average page walk latency by 37% under colocation. Meanwhile, prefetching from PL1 together with PL2 in both guest and host under workload colocation reduces page walk latency by an average of 45%. The best-case improvement of 55% is registered on `memcached` with 400GB dataset, whose average page walk latency drops by 378 cycles.

5.3 Estimation of Performance Improvement

In this section, we estimate performance improvement of ASAP which prefetches PL1 and PL2 in both guest and host when running in isolation under virtualization. We (1) quantify the fraction of cycles spent in page walks *on the critical path*, and (2) obtain a conservative estimate of ASAP's performance improvement by multiplying this fraction with ASAP's average reduction in page walk latency (see Figure 10a).

To quantify the fraction of cycles in page walks on the critical path, we measure execution time in the absence of TLB misses (hence, no page walks) and compare that to normal execution with TLB misses. To eliminate TLB misses, we run the applications using

Table 6: Conservative projection of ASAP's performance improvement.

	mcf	canneal	bfs	pagerank	redis	Average
Fraction of cycles spent in page walks on the critical path	31%	24%	68%	50%	18%	34%
ASAP's reduction in average page walk latency	25%	32%	41%	43%	33%	39%
ASAP's minimum performance improvement	8%	8%	28%	22%	6%	12%

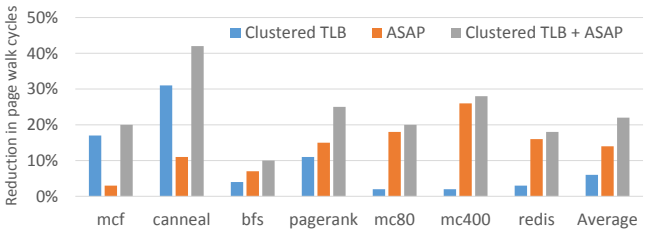


Figure 11: Reduction in the number of CPU cycles spent in page walks for Clustered TLB, ASAP, and the two together. Native execution in isolation (higher is better).

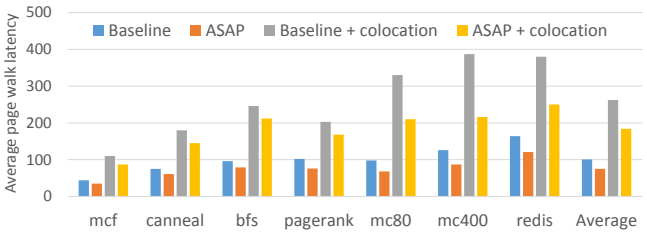


Figure 12: Average page walk latency with virtualization when hypervisor uses 2MB pages (lower is better). Baseline corresponds to execution in isolation.

Table 7: Reduction in TLB MPKI with clustered TLB. The data is normalized to native execution in isolation.

mcf	canneal	bfs	pagerank	mc80	mc400	redis	Average
58%	48%	10%	16%	4%	9%	12%	15%

a small (3GB or smaller) dataset while forcing an application to use large pages with *libhugetlbfs* [40]. With such a small dataset and large pages enabled, the capacity of L2 S-TLB (1536 entries) is enough to capture the whole page table. As a result, we achieve a significant (~100×) reduction in the number of page walks. The reduction in execution time due to page walks elimination corresponds to page walk cycles on the critical path. Note that using large pages can significantly reduce the number of page walks *only* for datasets smaller than 3GB (reach of the TLB). This and other limitations of large pages (see Section 2.3) make their use in a datacenter problematic.

We study all the applications except `memcached`, which is unaffected by *libhugetlbfs*. The results of the study are shown in Table 6. With page walks eliminated, the largest reduction in total execution time compared to a configuration where page walks are present is observed on graph workloads – 68% on `bfs` and 50% on `pagerank`. Projecting these results on ASAP, which in isolation under virtualization reduces average page walk latency by 41% on `bfs` (43% `pagerank`), ASAP improves performance by 28% (22% on `pagerank`). On average, ASAP is estimated to improve performance by 12%.

5.4 Comparison to Existing Techniques

We compare ASAP with state-of-the-art microarchitectural and software techniques and demonstrate their synergy with ASAP.

5.4.1 TLB Coalescing. TLB coalescing techniques [4, 5] detect and exploit available contiguity in virtual-to-physical mappings by coalescing TLB entries for adjacent pages into a single entry. Doing so increases effective TLB capacity and reduces TLB MPKI.

We evaluate Clustered TLB [5], a state-of-the-art TLB coalescing technique that coalesces up to 8 PTEs into 1 TLB entry. Table 7 shows the TLB MPKI reduction thanks to Clustered TLB. We find that Clustered TLB is highly effective for applications with smaller datasets, specifically `mcf` and `canneal`, reducing TLB MPKI by 58% and 48%, respectively. However, on the rest of the applications, which have much larger datasets (see Table 3), Clustered TLB is less effective, and TLB MPKI reduction varies from just 4% to 16%.

Figure 11 shows the reduction in page walk cycles with Clustered TLB, ASAP, and the two combined. Results are normalized to a baseline without either Clustered TLB or ASAP. On average, Clustered TLB reduces cycles spent in page walks by 5%, with largest improvement coming from workloads with small datasets. The reduction in the number of page walk cycles is smaller than reduction in TLB MPKI because the PT nodes accessed by page walks that are eliminated by Clustered TLB are the ones highly likely to be in higher-level caches due to spatio-temporal locality. Thus, clustered TLB eliminates mostly short page walks, leaving uncovered long page walks that access LLC and memory.

In contrast, ASAP is particularly effective in accelerating long page walks, particularly when both PL1 and PL2 nodes miss in higher-level caches. As a result, ASAP and clustered TLB naturally compliment each other and, when combined, can deliver additive performance gains. As shown in Figure 11, ASAP alone decreases the number of cycles spent in page walks by 14%, on average. Combining clustered TLB with ASAP increases TLB reach *and* reduces the walk latency, eliminating 22% of page walk cycles, on average, and 41% in the best case (on `canneal`).

5.4.2 ASAP with Large Pages. A common optimization employed by modern hypervisors under low to moderate memory pressure is allocating guest physical memory in large pages [19]. Doing so eliminates up to five long-latency accesses to the memory hierarchy on each walk (i.e., accesses 4, 9, 14, 19, 24 in Figure 7).

We evaluate ASAP with 2MB host pages, with prefetching from both PL1 and PL2 in the guest and PL2-only in the host. Figure 12 depicts the results for this study. The baseline corresponds to execution in isolation with host using 2MB pages. ASAP reduces page walk latency by 25%, on average, over the baseline, and by up to 31% in the best case (on `memcached` with 400GB dataset).

Under colocation, the average page walk latency increases by 2.6× as compared to execution in isolation. In this scenario, ASAP reduces page walk latency by 30%, on average, and by 44% in the best case on `memcached` with 400GB dataset, whose average page walk latency reduces by 171 cycles). Overall, we conclude that even with shortened page walks enabled by 2MB pages, ASAP delivers a considerable reduction in page walk latency.

6 RELATED WORK

Improving TLB reach. Prior art suggests a number of mechanisms to boost TLB's effective capacity by coalescing adjacent PT entries [4–6] or by sharing TLB capacity among CPU cores [41, 42] including the die-stacked L3 TLB design [16]. ASAP's advantage over the die-stacked L3 TLB is its microarchitectural simplicity and ability to work well under colocation. ASAP requires just a set of registers and simple comparison logic, whereas L3 TLB requires more than 16MB of die-stacked DRAM. Moreover, by heavily relying on the cache hierarchy, under colocation, L3 TLB is likely to suffer from thrashing and can experience increased miss rates. Thus, even L3 TLB would benefit from ASAP. Ultimately, TLB enhancements are constrained by a combination of area, power and latency. Given the continuing growth in dataset sizes, it is imperative to accelerate the latency of TLB misses, which is precisely the target of ASAP. As shown in Section 5.4.1, ASAP is complementary to techniques that coalesce adjacent PT entries.

TLB entries prefetching. Prior work explores a number of prefetch techniques to decrease the number of TLB misses. Kandiraju et al. study stride and markov TLB prefetchers that rely on available spatial and temporal locality of consecutive TLB misses [43]. Lustig et al. exploit inter-core prefetching that is efficient for the workloads that exhibit sufficient dataset sharing [42]. While these techniques mitigate the translation overheads for the workloads with regular memory access patterns, ASAP is oblivious to the TLB misses origin, decreasing the penalty of all the TLB misses including those induced by the irregular access patterns that are beyond the reach of TLB prefetchers.

Reducing page table access latency. To reduce PT access latency, prior work proposes replacing the PT radix tree with lookup-latency optimized data structures, such as hash tables [18, 44] and hashed inverted pagetables as in IBM PowerPC [45]. While promising, such designs are disruptive and may suffer from performance pathologies including long chain traversals due to hash collisions [17, 18]. SPARC architecture handles TLB misses in software while accelerating TLB miss handling by introducing a software-managed direct-mapped cache of translations, called TSB [46]. However, prior work shows that larger TSB entries exhibit poorer cache locality making TSB less efficient than the conventional PT radix tree [17].

Speculative address translation. SpecTLB [47] interpolates on existing TLB entries to predict translations when a reservation-based memory manager is used, as in FreeBSD [48, 49]. Thus, SpecTLB allows speculative execution of memory operations before their correctness is verified. This approach may pose security threats inherent to speculative execution of memory operations, as demonstrated by recent attacks such as Spectre [50], Meltdown [51], and Foreshadow [52]. In contrast, ASAP never consumes prefetched entries unless validated by a full page walk.

Translation-triggered prefetch. Bhattacharjee observes that if a page walker accesses main memory when servicing a TLB miss, the corresponding data is also likely to be memory-resident [53]. Hence, the author suggests enabling the memory controller to complete the translation in-place, so as to immediately prefetch the data for which the address translation is being carried out. This optimization can be seamlessly combined with ASAP, whose prefetches would reduce the latency of both the page walk and the data access.

Virtualization and nested page walks. Nested PTs introduce a significant performance overhead due to the elevated number of memory accesses in a page walk. Some researchers seek to limit the number of accesses by flattening the host PT [54], while the others use a unified PT structure, called shadow PT, managed by hypervisor [55]. Finally, Gandhi et al. combines nested and shadow PTs with a mechanism that dynamically switches between the two [1]. All of these techniques would benefit from ASAP, which would further reduce page walk latencies.

7 CONCLUSION

Existing techniques for lowering the latency of address translation without disrupting the established virtual memory abstraction all rely on caching – in TLBs, page walk caches and in the processor's memory hierarchy. Problematically, the trend toward larger application datasets, bigger machine memory capacities and workload consolidation means that these caching structures will be increasingly pressured by the need to keep an ever-larger number of translations. Thus, high page walk latencies due to frequent memory accesses are bound to become a "feature" of big-memory workloads.

This work takes a step toward lowering page walk latencies by prefetching page table entries in advance of demand accesses by the page walker, effectively uncovering memory level parallelism within a single page walk. This idea, which we call Address Translation with Prefetching (ASAP), is powered by an insight that the inherently serial radix tree traversal performed on a page walk can be accelerated through direct indexing into a given level of the page table. Such indexing can be achieved through a simple ordering of page table entries by the OS without modifications to the underlying page table structure. While ASAP does expose the latency of at least one access to the memory hierarchy, it is nonetheless highly effective, especially for virtualized and co-located workloads, reducing page walk latency by up to 55%. A strength of ASAP lies in the fact that it is a plug-and-play solution that works with the existing virtual memory abstraction and the full ensemble of today's address translation machinery.

ACKNOWLEDGMENTS

The authors thank Priyank Faldu, Adrien Ghosn, Marios Kogias, James Larus, George Prekas, Amna Shahab, as well as the members of EPFL's DCSL group for their valuable feedback and many fruitful discussions at the early stage of the work. This work was supported by the Google Faculty Research Award, the EPSRC Centre for Doctoral Training in Pervasive Parallelism at the University of Edinburgh, and the industrial CASE studentship from Arm Ltd.

REFERENCES

[1] J. Gandhi, M. D. Hill, and M. M. Swift, "Agile paging: Exceeding the best of nested and shadow paging," in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, 2016, pp. 707–718.

[2] Linley Group, "3D XPoint fetches data in a flash," *Microprocessor Report*, September 2015.

[3] Intel, "5-level paging and 5-level EPT," Intel, White Paper 335252-002, May 2017.

[4] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "CoLT: Coalesced large-reach TLBs," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012, pp. 258–269.

[5] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing TLB reach by exploiting clustering in page translations," in *Proceedings of the 20th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2014, pp. 558–567.

[6] C. H. Park, T. Heo, J. Jeong, and J. Huh, "Hybrid TLB coalescing: Improving TLB translation coverage under diverse fragmented memory allocations," in *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*, 2017, pp. 444–456.

[7] G. Cox and A. Bhattacharjee, "Efficient address translation for architectures with multiple page sizes," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXII)*, 2017, pp. 435–448.

[8] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, "Coordinated and efficient huge page management with ingens," in *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, 2016, pp. 705–721.

[9] B. Pham, J. Veselý, G. H. Loh, and A. Bhattacharjee, "Large pages and lightweight memory management in virtualized environments: Can you have it both ways?" in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 1–12.

[10] A. Seznec, "Concurrent support of multiple page sizes on a skewed associative TLB," *IEEE Trans. Computers*, vol. 53, no. 7, pp. 924–927, 2004.

[11] M.-M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos, "Prediction-based superpage-friendly TLB designs." in *Proceedings of the 21st IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2015, pp. 210–222.

[12] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, 2013, pp. 237–248.

[13] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "Efficient memory virtualization: Reducing dimensionality of nested page walks," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014, pp. 178–189.

[14] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal, "Redundant memory mappings for fast access to large memories," in *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, 2015, pp. 66–78.

[15] H. Alam, T. Zhang, M. Erez, and Y. Etsion, "Do-it-yourself virtual memory translation," in *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*, 2017, pp. 457–468.

[16] J. H. Ryoo, N. Gulur, S. Song, and L. K. John, "Rethinking TLB designs in virtualized environments: A very large part-of-memory TLB," in *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*, 2017, pp. 469–480.

[17] T. W. Barr, A. L. Cox, and S. Rixner, "Translation caching: skip, don't walk (the page table)," in *Proceedings of the 37th International Symposium on Computer Architecture (ISCA)*, 2010, pp. 48–59.

[18] I. Yaniv and D. Tsafrir, "Hash, don't cache (the page table)," in *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2016, pp. 337–350.

[19] E. Bugnion, J. Nieh, and D. Tsafrir, *Hardware and software support for virtualization*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2017.

[20] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating two-dimensional page walks for virtualized systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIII)*, 2008, pp. 26–35.

[21] J. Araujo, R. Matos, P. Maciel, R. Matias, and I. Beicker, "Experimental evaluation of software aging effects on the eucalyptus cloud computing infrastructure," in *Proceedings of the Middleware 2011 Industry Track Workshop*. ACM, 2011, p. 4.

[22] J. Hu, X. Bai, S. Sha, Y. Luo, X. Wang, and Z. Wang, "HUB: Hugepage ballooning in kernel-based virtual machines," in *Proceedings of the 4th International Symposium on Memory Systems (MEMSYS)*, 2018.

[23] K. Keeton, "Memory-driven computing," in *Proceedings of 15th USENIX Conference on File and Storage Technologies (FAST)*, 2017.

[24] D. Tang, P. Carruthers, Z. Totari, and M. W. Shapiro, "Assessment of the effect of memory page retirement on system RAS against hardware faults," in *Proceedings of the 36th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2006, pp. 365–370.

[25] "Bad page offlining," 2009. [Online]. Available: www.mcelog.org/badpageofflining.html

[26] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field," in *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2015, pp. 415–426.

[27] L. Zhang, B. Neely, D. Franklin, D. B. Strukov, Y. Xie, and F. T. Chong, "Mellow writes: Extending lifetime in resistive memories through selective slow write backs," in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, 2016, pp. 519–531.

[28] M. Zhang, L. Zhang, L. Jiang, Z. Liu, and F. T. Chong, "Balancing performance and lifetime of MLC PCM by using a region retention monitor," in *Proceedings of the 23rd IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2017, pp. 385–396.

[29] Y. Du, M. Zhou, B. R. Childers, D. Mossé, and R. G. Melhem, "Supporting superpages in non-contiguous physical memory," in *Proceedings of the 21st IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2015, pp. 223–234.

[30] S. Sivaram, "Storage class memory: Learning from 3D NAND," Flash Memory Summit, 2016. [Online]. Available: www.flashmemorysummit.com/English/Collaterals/Proceedings/2016/20160809_Keynote4_WD_Sivaram.pdf

[31] K. Suzuki and S. Swanson, "The non-volatile memory technology database (NVMDB)." University of California, San Diego, Tech. Rep. CS2015-1011, 2015.

[32] R. Kath, "Managing virtual memory," 1993. [Online]. Available: msdn.microsoft.com/en-us/library/ms810627.aspx

[33] A. Arcangeli, "Transparent hugepage support," *KVMForum*, 2010.

[34] "Galois," 2018. [Online]. Available: iss.ices.utexas.edu/?p=projects/galois

[35] A. Bhattacharjee, "Large-reach memory management unit caches," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2013, pp. 383–394.

[36] D. Bruening and S. Amarasinghe, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2004.

[37] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*, 2015, pp. 1–17.

[38] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, 2015, pp. 158–169.

[39] S. Van Schaik, K. Razavi, B. Gras, H. Bos, and C. Giuffrida, "RevAnC: A framework for reverse engineering hardware page table caches," in *Proceedings of the 10th European Workshop on Systems Security*. ACM, 2017, p. 3.

[40] "libhugetlbfs(7) - Linux man page," 2006. [Online]. Available: linux.die.net/man/7/libhugetlbfs

[41] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level TLBs for chip multiprocessors," in *Proceedings of the 17th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2011, pp. 62–63.

[42] D. Lustig, A. Bhattacharjee, and M. Martonosi, "TLB improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level TLBs," *TACO*, vol. 10, no. 1, pp. 2:1–2:38, 2013.

[43] G. B. Kandiraju and A. Sivasubramaniam, "Going the distance for TLB prefetching: An application-driven study," in *Proceedings of the 29th International Symposium on Computer Architecture (ISCA)*, 2002, pp. 195–206.

[44] Intel, "Intel Itanium© architecture software developer's manual, Volume 2," 2010. [Online]. Available: www.intel.com/content/www/us/en/processors/itanium/itanium-architecture-software-developer-rev-2-3-vol-2-manual.html

[45] IBM, "Power ISA version 2.07 B," 2018. [Online]. Available: https://openpowerfoundation.org/?resource_lib=ibm-power-isa-version-2-07-b

[46] Sun Microsystems, "UltraSPARC T2 supplement to the UltraSPARC architecture," 2007. [Online]. Available: https://www.oracle.com/technetwork/systems/opensparc/t2-13-ust2-uasuppl-draft-p-ext-1537760.html

[47] T. W. Barr, A. L. Cox, and S. Rixner, "SpecTLB: A mechanism for speculative address translation," in *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*, 2011, pp. 307–318.

[48] J. Navarro, S. Iyer, P. Druschel, and A. L. Cox, "Practical, transparent operating system support for superpages," in *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI)*, 2002.

[49] M. Talluri and M. D. Hill, "Surpassing the TLB performance of superpages with less operating system support," in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, 1994, pp. 171–182.

[50] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 19–37.

[51] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *Proceedings of the 27th USENIX Security Symposium*,

2018, pp. 973–990.

[52] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *Proceedings of the 27th USENIX Security Symposium*, 2018, pp. 991–1008.

[53] A. Bhattacharjee, "Translation-triggered prefetching," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXII)*, 2017, pp. 63–76.

[54] J. Ahn, S. Jin, and J. Huh, "Revisiting hardware-assisted page walks for virtualized systems," in *Proceedings of the 39th International Symposium on Computer Architecture (ISCA)*, 2012, pp. 476–487.

[55] C. Waldspurger, "Memory resource management in VMware ESX server," *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.